THE UNIVERSITY of TEXAS
HEALTH SCIENCE CENTER AT HOUSTON
SCHOOL of HEALTH INFORMATION SCIENCES

Introduction to C++ Part II

For students of HI 5323
"Image Processing"

Willy Wriggers, Ph.D.
School of Health Information Sciences

http://biomachina.org/courses/processing/03.html

Review of Last Session
by Example

```
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

```
Welcome to C++!
```

```cpp
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

Comments

Written between `/*` and `*/` or following a `//`.

Improve program readability and do not cause the computer to perform any action.

```
Welcome to C++!
```

```cpp
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that progr
}
```

preprocessor directive

Message to the C++ preprocessor.

Lines beginning with **#** are preprocessor directives.

#include <iostream> tells the preprocessor to include the contents of the file **<iostream>**, which includes input/output operations (such as printing to the screen).

```
Welcome to C++!
```

```cpp
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

```
Welcome to C++!
```

C++ programs contain one or more functions, one of which must be **`main`**

Parenthesis are used to indicate a function

`int` means that **`main`** "returns" an integer value. More in Chapter 3.

A left brace **`{`** begins the body of every function and a right brace **`}`** ends it.

```cpp
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

```
Welcome to C++!
```

Prints the *string* of characters contained between the quotation marks.

The entire line, including **`std::cout`**, the **`<<`** *operator*, the *string* **`"Welcome to C++!\n"`** and the *semicolon* (**`;`**), is called a *statement*.

All statements must end with a semicolon.

```cpp
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream>

int main()
{
   std::cout << "Welcome to C++!\n";

   return 0;      // indicate that program ended successfully
}
```

```
Welcome to C++!
```

`return` is a way to exit a function from a function.

`return 0`, in this case, means that the program terminated normally.

```cpp
// Fig. 1.4: fig01_04.cpp
// Printing a line with multiple statements
#include <iostream>

int main()
{
   std::cout << "Welcome ";
   std::cout << "to C++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome to C++!
```

Unless new line `'\n'` is specified, the text continues on the same line.

```cpp
// Fig. 1.5: fig01_05.cpp
// Printing multiple lines with a single statement
#include <iostream>

int main()
{
   std::cout << "Welcome\nto\n\nC++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome
to

C++!
```

Multiple lines can be printed with one statement.

```cpp
// Fig. 1.6: fig01_06.cpp
// Addition program
#include <iostream>

int main()
{
   int integer1, integer2, sum;           // declaration

   std::cout << "Enter first integer\n";  // prompt
   std::cin >> integer1;                  // read an integer
   std::cout << "Enter second integer\n"; // prompt
   std::cin >> integer2;                  // read an integer
   sum = integer1 + integer2;             // assignment of sum
   std::cout << "Sum is " << sum << std::endl; // print sum

   return 0;   // indicate that program ended successfully
}
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

```cpp
// Fig. 1.6: fig01_06.cpp
// Addition program
#include <iostream>

int main()
{
   int integer1, integer2, sum;           // declaration

   std::cout << "Enter first integer\n";  // p
   std::cin >> integer1;                  // r
   std::cout << "Enter second integer\n"; // prompt
   std::cin >> integer2;                  // read an integer
   sum = integer1 + integer2;             // assignment of sum
   std::cout << "Sum is " << sum << std::endl; // print sum

   return 0;   // indicate that program ended success
}
```

Notice how **`std::cin`** is used to get user input.

`std::endl` flushes the buffer and prints a newline.

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Variables can be output using **`std::cout << variableName`**.

```
// Fig. 1.14: fig01_14.cpp
// Using if statements, relational
// operators, and equality operators
#include <iostream>

using std::cout;  // program uses cout
using std::cin;   // program uses cin
using std::endl;  // program uses endl

int main()
{
   int num1, num2;

   cout << "Enter two integers, and I will tell you\n"
        << "the relationships they satisfy: ";
   cin >> num1 >> num2;   // read two integers

   if ( num1 == num2 )
      cout << num1 << " is equal to " << num2 << endl;

   if ( num1 != num2 )
      cout << num1 << " is not equal to " << num2 << endl;

   if ( num1 < num2 )
      cout << num1 << " is less than " << num2 << endl;

   if ( num1 > num2 )
      cout << num1 << " is greater than " << num2 << endl;

   if ( num1 <= num2 )
      cout << num1 << " is less than or equal to "
           << num2 << endl;

```

```cpp
// Fig. 1.14: fig01_14.cpp
// Using if statements, relational
// operators, and equality operators
#include <iostream>

using std::cout;  // program uses cout
using std::cin;   // program uses cin
using std::endl;  // program uses endl

int main()
{
   int num1, num2;

   cout << "Enter two integers, and I will tell you\n"
        << "the relationships they satisfy: ";
   cin >> num1 >> num2;   // read two integers

   if ( num1 == num2 )
      cout << num1 << " is equal to " << num2 << endl;

   if ( num1 != num2 )
      cout << num1 << " is not equal to " << num2 << endl;

   if ( num1 < num2 )
      cout << num1 << " is less than " << num2 << endl;

   if ( num1 > num2 )
      cout << num1 << " is greater than " << num2 << endl;

   if ( num1 <= num2 )
      cout << num1 << " is less than or equal to "
           << num2 << endl;

```

Notice the **`using`** statements.

The **`if`** statements test the truth of the condition. If it is **`true`**, body of **`if`** statement is executed. If not, body is skipped.

To include multiple statements in a body, delineate them with braces **`{}`**.

```
// Fig. 1.14: fig01_14.cpp
// Using if statements, relational
// operators, and equality operators
#include <iostream>

using std::cout;  // program uses cout
using std::cin;   // program uses cin
using std::endl;  // program uses endl

int main()
{
   int num1, num2;

   cout << "Enter two integers, and I will tell you\n"
        << "the relationships they satisfy: ";
   cin >> num1 >> num2;   // read

   if ( num1 == num2 )
      cout << num1 << " is equal to " << num2 << endl;

   if ( num1 != num2 )
      cout << num1 << " is not equal to " << num2 << endl;

   if ( num1 < num2 )
      cout << num1 << " is less than " << num2 << endl;

   if ( num1 > num2 )
      cout << num1 << " is greater than " << num2 << endl;

   if ( num1 <= num2 )
      cout << num1 << " is less than or equal to "
           << num2 << endl;

```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
```

```
3 is not equal to 7
```

```
3 is less than 7
```

```
3 is less than or equal to 7
```

```
   if ( num1 >= num2 )
      cout << num1 << " is greater than or equal to "
           << num2 << endl;

   return 0;   // indicate that program ended successfully
}
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Functions and Memory Concepts

Memory Concepts

- Variables
  - Correspond to locations in the computer's memory
  - Every variable has a name, a type, a size and a value
  - Whenever a new value is placed into a variable, it replaces the previous value - it is destroyed
  - Reading variables from memory does not change them
- A visual representation:

integer1 | **45**

Pointer

- A pointer is a value that denotes an object location in memory. A pointer variable is a variable that holds pointer values. The type associated with a pointer variable or value constrains the kind of object or variable at the designated location.

Reference

- A reference is an alternative name for an object. The notation &x means reference to x. A reference must be initialized.

```
void f()
{
  int i = 1;
  int& r = i; // r and i refer to same int
  int x = r; // x = 1
  r = 2; // i = 2
}
```

Examples

```
int  x, y;
int *p  = &x;   //  p holds the location of an integer variable.
int &q = y;     //  q  holds a reference of an integer variable.
                //  q and y are now names of the same variable!

int  *f() { return &x; }   // result value points to x.

int  &g(){ return x; }     // result value can substitute for a reference to x.

void swap( int *a, int *b){  int t = *a;  *a = *b; *b = t; }  //call:  swap(&x, &y);

void swap( int &a, int &b){ int t = a; a = b; b = t; }  //call: swap(x, y);
```

Color: meaning of * and & symbols depend on the context:

- In a declaration, int *x means x is a pointer to a variable of type int
- In a statement, *x means the actual variable pointed to by pointer x
- In a declaration, int &x means x holds the name of a variable of type int
- In a statement, &x means the location of variable x

Examples

Color: meaning of * and & symbols depend on the context:

- In a declaration, int *x means x is a pointer to a variable of type int
- In a statement, *x means the actual variable pointed to by pointer x
- In a statement, &x means the location of variable x

Location	Variable
x	*x
&x	x

Examples

Color: meaning of * and & symbols depend on the context:

- In a declaration, int *x means x is a pointer to a variable of type int
- In a statement, *x means the actual variable pointed to by pointer x
- In a statement, &x means the location of variable x

Location	Variable
x	*x
&x	x

intuitive (C-style): * and & change between location and variable:

- (int *)x: x has “pointer to int” type
- int (*x): *x has int type, is actual variable
- & is antidote to * in statements

Examples

Color: meaning of * and & symbols depend on the context:

- In a declaration, int *x means x is a pointer to a variable of type int
- In a statement, *x means the actual variable pointed to by pointer x
- In a statement, &x means the location of variable x

Location	Variable
x	*x
&x	x

intuitive (C-style): * and & change between location and variable:

•(int *)x: x has “pointer to int” type

•int (*x): *x has int type, is actual variable

•& is antidote to * in statements

- In a C++ declaration, int &x means x holds the name of a variable of type int

careful, there is no “location-variable type change” when used this way

Argument Passing

void foo(int x); // pass by value

void foo(int* x); // pass using pointer

void foo(int &x); // pass by reference (new)

Argument Passing

```
void f(int val, int &ref)
{
  val++;
  ref++;
}
```

- When f() is called, val++ increments a local copy of the 1^{st} argument, whereas ref++ increments the 2^{nd} argument.

Argument Passing

```
void g()
{
  int i = 1;
  int j = 1;
  f(i, j); // increments j but not i
}
```

- The 1st argument, i, is passed by value, the 2nd argument, j, is passed by reference.

Argument Passing

- It can be more efficient to pass a large object by reference than to pass it by value.
- Declaring an argument const does not enable the called function to change the object value.

```
void f(const Large& arg)
{
  // the value of arg cannot be changed
}
```

Inline Functions

- A direction for the preprocessor to substitute code

C macro:

```
#define max(a,b) (((a) > (b) ? (a) :
  (b))
```

C++ inline function:

```
inline int max(int a, int b) {return
  a>b ? a : b ;}
```

Inline Functions

- Problems with macros in C
  - Can be a source of problems
  - Has no class scope !!
- C++ solves the problem with inline functions
  - Under the control of the compiler
  - Expanded in-place

Memory Management in C++

new and delete operators

```
void foo()
{
        int *p = new int;

        delete p;
}
```

```
void foo()
{
        int *p = new int[23];

        delete [] p;
}
```

Memory Management in C++

- Do not mix array and non-array allocations.

```
void foo()
{
        int* p = new int[100];

        delete p;  // disaster!
}
```

Memory Management in C++

• Do not mix C-style memory management with C++ memory managment.

```
void foo()
{
        int* p = new int;

        free(p);  // disaster!
}
```

• Do not use malloc and free.

Overloaded Function Names

- Using the same name for operations on different types is called *overloading*.

```
int max(int, int);
double max(double, double);
long max(long, long);
```

Overloaded Function Names

- Finding the right version to call from a set of overloaded functions is done by looking for a best match between the type of the argument expression and the parameters of the functions:
  1. Exact match
  2. Match using promotions: bool, char, short to int; float to double.
  3. Match using standard conversions:
     int to double, double to int

Example

```
void print(char);
void print(int);

void h(char c, int i, short s, double d)
{
  print(c); // exact match: invoke print(char)
  print(i); // exact match: invoke print(int)
  print(s); // promotion: invoke print(int)
  print(d); // conversion: double to int
}
```

Overloaded Operators

- enable conventional notations

```
Inline bool operator==(Date a,Date b) //equality
{       return a.day()==b.day() &&
a.month()==b.month() && a.year()==b.year();
}
bool operator!=(Date,Date);      //inequality;
bool operator<(Date,Date);       //less than
bool operator>(Date,Date);        //greater than
```

Default Arguments

- Often a function needs more arguments than necessary to handle simple cases.
- Default values may be provided for trailing arguments only:

```
void foo (int a, int b = 0)
   int a, b;
   foo (a, b);
   foo (a); // called foo function with
            // arguments (a, 0)
```

Classes

Classes and Data Abstraction

- C style structure

```
struct point {
    int x;
    int y;
};

struct point {
    unsigned int r;
    unsigned int theta;
};
```

- C++ class

```
class point {
public:
    point (void);
    ~point (void);
    int    getX (void) const;
    int    getY (void) const;
    unsigned int getR (void) const;
    unsigned int getTheta (void) const;
private:
    int x_;
    int y_;
};
```

A class definition is in a header file: .h file

A class implementation is in a .cc, .cpp, .cxx file

Object-Oriented Programming

- Encapsulation of a set of data types and their operations: the *class* construct.
- Data hiding.
- Data type hierarchy & code reuse via inheritance: deriving a new class from an existing one.
- Design is crucial!

Classes

- A *class* is a user-defined type, which allows encapsulation.
- The construct

  class X { ... };

  is a class definition.
- Contains data and function members.
- Access control mechanisms (private vs. public – see below).

Example

```
class Point {
private:
    int x, y, visible; // private part
                       // can be used only by member functions
public: // public part, interface
    void setpoint(int xx, int yy) { x=xx; y=yy; visible = 1; }
    int getx() { return x; }
    int gety() { return y; }
    void draw() {
      gotoxy(x,y);
      putchar('*');
      visible = 1;
    }
    void hide() {
      gotoxy(x,y);
      putchar(' ');
      visible = 0;
    }
    // member functions defined within the class definition,
};  // rather than simply declared there are inline.
```

Example

```
Point p1, p2;
p1.setpoint(10,20);
p1.draw();
p2.setpoint(15,15);
p2.draw();
p1.hide();
```

Member Functions

- A member function declaration specifies:

  1. The function can access the private part of the class declaration.
  2. The function is in the scope of the class.
  3. The function must be invoked on an object.

- In a member function, member names can be used without explicit reference to an object.

Example

```
class Array {
private:
  int *parray;
  int size;
public:
  void init();
  int get(int indx);
  void print();
  void set(int indx, int value);
};
// :: is the scope resolution operator
void Array::init(){ parray = 0; size = 0; }
int Array::get(int i) { return parray[i]; }
void Array::print()
{
  for (int i = 0; i < size; i++)
      cout << endl << “array[“ << i << “]=“ <<
      parray[i];
}
```

Example

```
void Array::set(int indx, int value)
{
  if (indx > size) {
      int *p = new int[indx+1];
      for (int i = 0; i < size; i++)
            p[i] = parray[i];
      delete [] parray;
      size = indx;
      parray = p;
  }
  parray[indx] = value;
}

Array a1;
a1.init();
a1.set(3,50);
a1.set(1,100);
a1.set(2,70);
a1.print();
```

Constructors

- Using functions such as init() to initialize class objects is error prone and complicates the code.
- Constructors are member functions with the explicit purpose of constructing values of a given type, recognized by having the same name as the class itself.

Destructors

- A constructor initializes an object, creating the environment in which the member functions operate. This may involve acquiring a resource such as a file, lock, and usually memory, that must be released after use.
- Destructor is a function that is guaranteed to be invoked when an object is destroyed, cleans up and releases resources.

Constructors & Destructors

```
class Array {
    Array();        // default constructor
    Array(int);     // constructor
    ~Array();       // destructor
};

Array::Array() {
    parray = 0;
    size = 0;
}
Array::Array(int len) {
    parray = new int[len];
    size = len;
}
Array::~Array() {
    delete [] parray;
}
```

Copy Constructor

- If x is an object of class X, “X y=x;” (or, equivalently “X y(x);”) by default means member-wise copy of x into y. This can cause undesired effects when used on objects of a class with pointer members.
- The programmer can define a suitable meaning for copy operations by a copy constructor (and similarly for the assignment operator).

```
Table::Table(const Table& t) {
  p = new Name[size = t.size];
  for (int i = 0; i < size; i++) p[i] = t.p[i];
}
String::String(const String &s) {
  str = new char[s.length()+1];
  strcpy(str, s);
}
```

Self-Reference

- *This* points to the object for which a member function is invoked.
- Used to return the object or to manipulate a self-referential data structure.

```
Date& Date::set_date(int dd, int mm, int yy)
{
    d = dd;
    m = mm;
    y = yy;
    return *this; // enables cascading
}

d.set_date( 20, 1, 2000 ).print();
```

Friends

- A friend function declaration specifies: the function can access the private part of the class declaration.
- Useful for object output (see below).

Example

```
class Matrix;
class Vector {
     int v[4];
public:
     Vector(){ v[0] = v[1] = v[2] = v[3] = 0; }
     int& elem(int i) { return v[i]; }
     friend Vector multiply(const Matrix &, const Vector &);
};
class Matrix {
     Vector v[4];
public:
     int& elem(int i, int j) { return v[i].elem(j); }
     friend Vector multiply(const Matrix &, const Vector &);
};
// multiply can be a friend of both Matrix and Vector
Vector multiply(const Matrix & m, const Vector & v) {
     Vector r;
     for (int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++)
               r.v[i] += m.v[i].v[j] * v.v[j];
     return r;
};
```

Derived Classes: Inheritance

```
employee:            manager:
     name                 name
     age                  age
     …                    …
                          group
                          level
```

Derived Classes

```
// a manager is an employee
// derived
class manager : public employee {
    employee* group; // people managed
    short level;
};
```

$$\begin{array}{c}\texttt{employee}\\ \uparrow\\ \texttt{manager}\end{array}$$

- Now we can put managers onto a list of employees without writing special code for managers.

Access Control

- A member of a class can be private, protected, or public.

1. If it's *private*, its name can be used only by member functions and friends of the class in which it is declared.

2. If it's *protected*, its name can be used only by member functions and friends of the class in which its declared and by member function and friends of classes derived from this class.

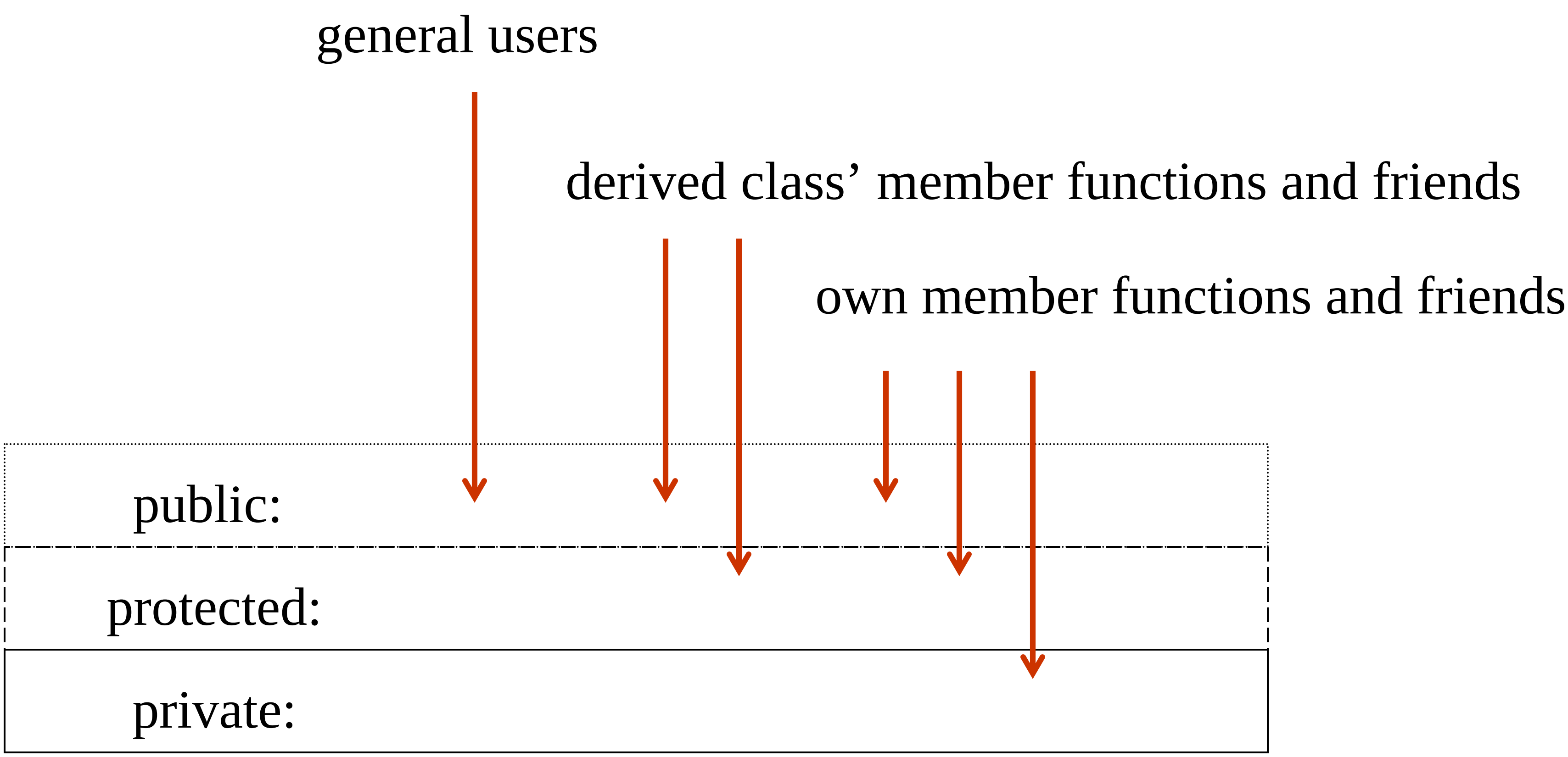
3. If it's *public*, its name can be used by any function.

Access Control

```
class Base1 {
private:
    int i;
protected:
    int j;
public:
    int k;
};

main () {
  Base1 b;
  int x;
  x = b.i; // error
  x = b.j; // error
  x = b.k; // ok
}
```

Access Control



Class Hierarchies

- A derived class can itself be a base class.
- A class may be derived from any number of base class.

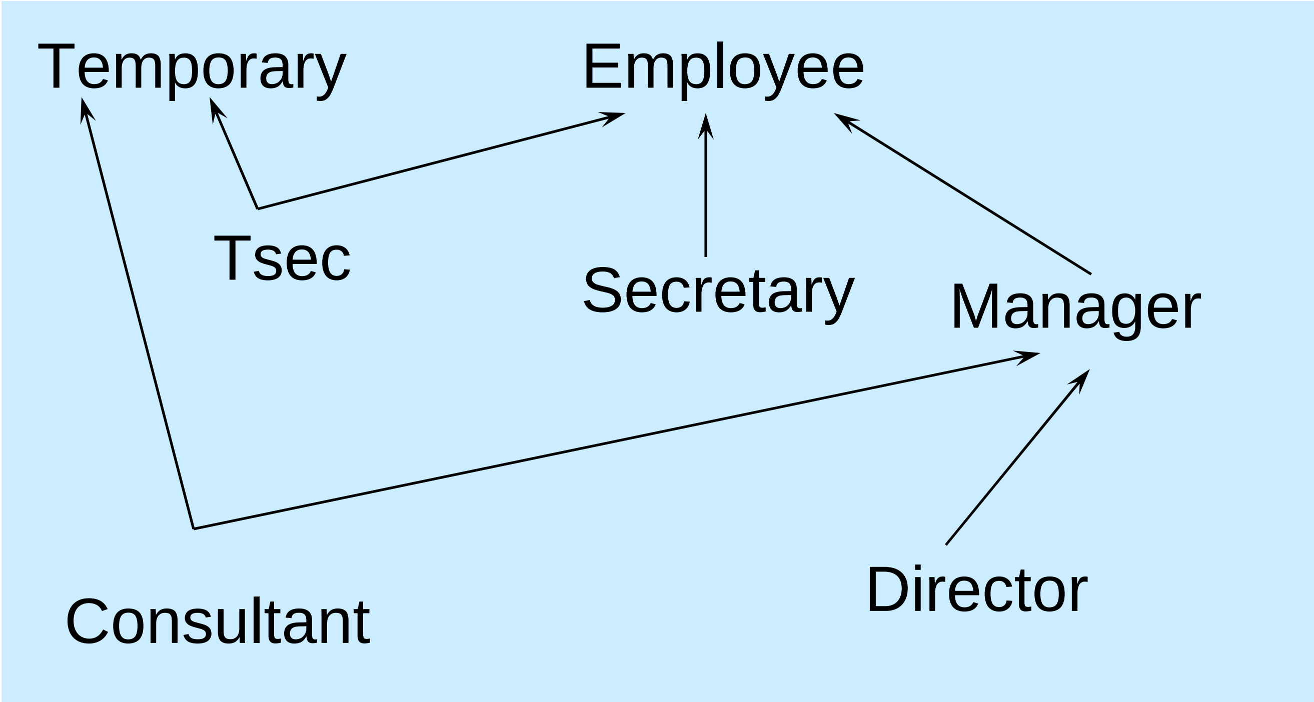
  -> multiple inheritance

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };

class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };

class Tsec  : public Temporary, public Employee { /* ... */ };
class Consultant
   : public Temporary, public Manager { /* ... */ };
```

Class Hierarchies



Class Hierarchy

Virtual Functions

- Virtual function allows the programmer to declare functions in base class that can be redefined in each derived class

```
Class Employee {
   string first_name, family_name;
public :
   Employee(const string& name, int dept);
   virtual void print( ) const;
   // . . .
};
void Employee::print( ) const
{
   cout << family_name<<'\n';
   // . . .
}
```

The key word virtual indicates that print( ) can act as interface to the print( ) defined in this class and the print( ) defined in classes derived from it

Virtual Functions

Derived class

```
struct Manager : public Employee {
    set <Employee*> group;
    short     level;
    // …
public :
    Manager(const string& name, int dept, int lvl);
    void print() const;
};

void Manager::print( ) const
{
    Employee::print( );
    cout << “\tlevel” << level << ‘\n’;
    // . . .
}
```

A function from a derived class with the same name and the same set of argument types as a virtual function in base class said to *override* the base class version of the virtual function

Abstract Classes

- A class with one or more pure virtual function is an abstract class.
  - No objects of the abstract class can be created
  - An abstract class can be used only as a base class of some other class
- An abstract class mechanism supports a general concept that links related objects.

Abstract Classes

```
class Shape {                    // abstract class
public:
   virtual void rotate(int) = 0;   // pure virtual functions
   virtual void draw() = 0;        // pure virtual functions
   virtual is_losed() = 0; // pure virtual functions
   // ...
};

Shape s;    // error : variable of abstract class Shape

class Circle : public Shape {
public:
        void rotate(int) {  }     // override Shape::rotate
        void draw( );              // override Shape::draw
        bool is_closed( ) { return true; } // override Shape::is_closed
        Circle(Point p, int r);
};
```

Exceptions

- Exceptions are special classes used for *error handling at runtime.*
- Handle errors at appropriate level
  - Exception propagates up call stack until "caught".
- Convey useful information to handler
  - Error class
  - Error details

Defining Exceptions

- Just another class…

```
class base_error
  {
    public:
    const char * s;
    error (const char * _s = 0);
    virtual const char * what () { return s; }
  };

  class range_error : public base_error
  {
    public:
    const double x;
    range_error (double _x) : x(_x) {}
    const char * what () { … // construct and return message }
  };

  class system_error : public base_error
  {
    public:
    const int errno;
    system_error (int _errno, const char * s = 0) : errno(_errno) {}
    const char * what () { … // return system error message }
  };
}
```

Throwing Exceptions

- Raise an exception by using the keyword “throw”

```
double average_grades (istream & in)
{
  double  x;
  double  S = 0;
  int     n = 0;

  while (in >> x)
  {
    if (x < 0 || x > 100)
      throw range_error (x);
    S += x;
    n++;
  }

  if (n == 0) throw base_error (“empty input file”);

  return S / n;
}
```

Catching Exceptions

- Catch an exception via a “try..catch” block

```
int main (int, char *[])
{
  using namespace std;

  try
  {
    cout << “average: “ << average_grades(cin) << endl;
  }

  catch (range_error & e)
  {
    cerr << “encountered a peculiar grade: ” << e.x << endl;
    exit (1);
  }

  catch (error & e)
  {
    cerr << “unable to calculate averages: “ << e.what() << endl;
    exit (2);
  }

  return 0;
}
```

Static (Global) Class Members

static *[* const | volatile *]* type data_member;

- Shared by all class instances
- One instance allocated in static data area and initialized at load time, or prior to execution of main().
- MUST NOT be initialized by a constructor or modified by a destructor; these methods manipulate non-static data members.
- MUST be initialized ONE TIME in (.cpp)(class implementation) file as follows

[const | volatile *]* type class_name ::data_member = initialization_expression;

Static Methods

static type method(…);

- Allow access to static data members!
- Do not have a this pointer; cannot access non-static data members!
- Called by the statement:

classname :: method(...);

- Can be called via class instances like other methods.
- Can be public, protected, or private.
- Can be in-line.
- Cannot be const or volatile.

Standard Library & Namespaces

Standard Library

- Containers
- Iterators
- Algorithms
- Diagnostics
- Strings
- I/O
- Localization
- Language support
- Numerics / Math

Hello, World!

- The line *#include<iostream>* instructs the compiler to include the declarations of the standard stream I/O facilities as found in *iostream*

```
#include <iostream>
int main( )
{
 std::cout << "Hello, world!\n";
}
```

The standard library is defined in a namespace called *std*

Namespaces

- Explicitly partition globally-scoped type definitions and variable names into logical segments
    - avoid name conflicts among multiple libraries, files, etc

```
namespace my
{
  const double version = 1.1;
  class string {…};
  class vector {…};
}

namespace your
{
  const int version = 3;
  class string {…};
  class vector {…};
}
```

Using Namespaces

- Import namespace into scope

```
using namespace my;
string s; // implicitly use my::string
```

- Import individual symbols into scope

```
use my::string;
use your::vector;

string s; // my::string
vector v; // your::vector
```

- Explicitly qualify symbols

```
my::string s;
your::vector v;
```

The Standard Library Namespace

- Every standard library is provided through some standard header
  - *#include<string>*
  - *#include<list>*
- To use them, the *std::* prefix can be used
  - std::string s = “Four legs Good; two legs Baaad!”;
  - std::list<std::string> slogans;

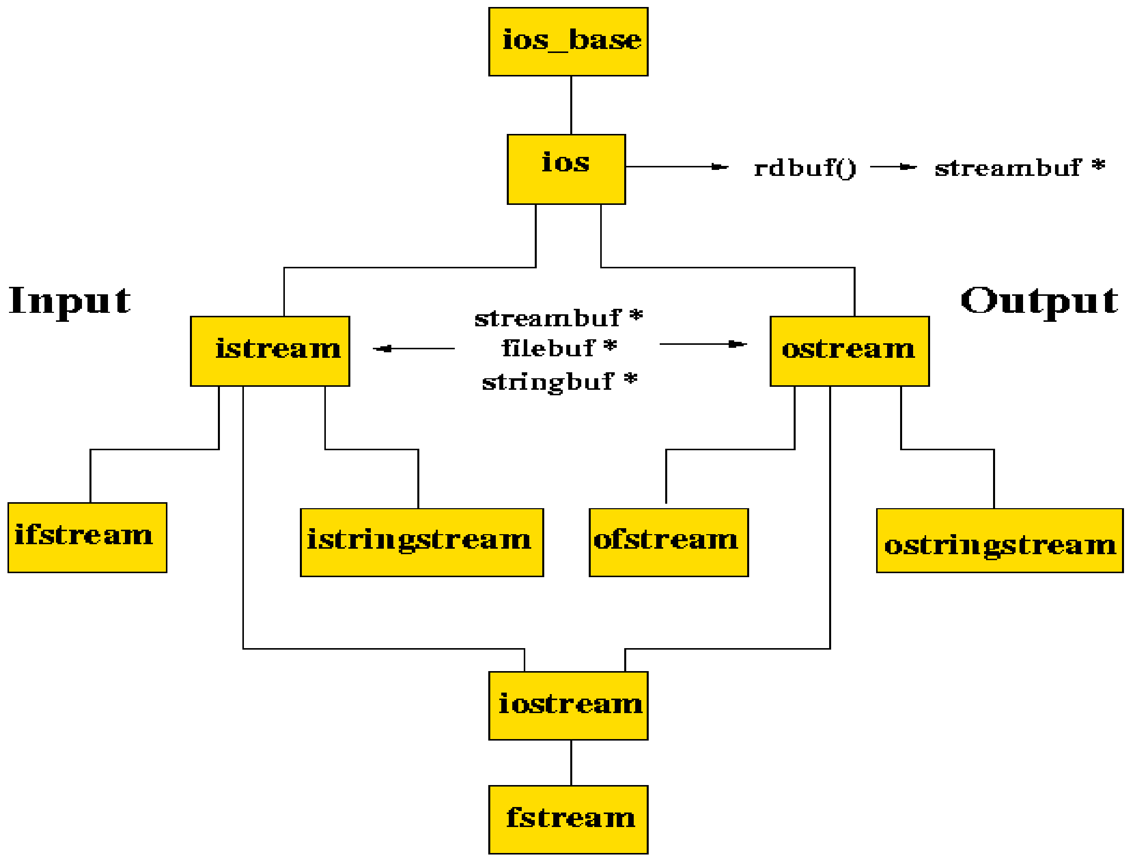
C++ IO: Input

- C++ input based on istream class
  - istream offers basic input functionalities
  - ifstream (used for file input)
  - istringstream (input from text not stored in files)
- cin is an object/instance of istream class (already defined when including <iostream>

C++ IO: Output

- C++ output based on ostream class
  - ostream offers basic output functionalities
  - ofstream (used for file output)
  - ostringstream (output to memory)
- cout is an object/instance of ostream class (already defined when including <iostream>

C++ Streams



ios_base
ios
rdbuf()
streambuf *
Input
Output
istream
streambuf *
filebuf *
stringbuf *
ostream
ifstream
istringstream
ofstream
ostringstream
iostream
fstream

File IO

```
#include <iostream>
#include <fstream>
using namespace std;
const int  cutoff = 6000;
const float rate1 = 0.3;
const float rate2 = 0.6;
int main() {
     // file object declarations
    ifstream fin;                    // ifstream is a subclass of fstream for input streams
    fin.open("income.dat“, ios::in);   // external file name specified in open method
    ofstream fout;                   // ofstream is a subclass of fstream for output streams
    fout.open("tax.out“: ios::out);
    int  income; float tax;      // declarations are compiled and have their effect as encountered
    while ( fin >> income ){       // equates to "true" until EOF is encountered (fin != 0)
         if( income < ::cutoff )     //  ::cutoff refers to the global name “cutoff”
           tax = ::rate1*income;   // implicit type conversion between int and float
         else
           tax = ::rate2*income;
        fout << "Income = " << income << " Drachma \n"
             <<  "      Tax: " << (int) tax*100.0 << " Lepta" << endl;

     }//while
    fin.close();                     // close file objects
    fout.close();
    return 0;
}
```

File IO

```
#include <fstream>
#include <string>
using namespace std;
int main() {
    // file object declarations
    double rate1 = 0.23, rate2 = 0.37, cutoff = 50000.00;
    string  fin_name, fout_name;
    cout << “Enter name of input file: “;    cin >> fin_name;    cout << endl;
    cout << “Enter name of output file: “;  cin >> fout_name;  cout << endl;
    ifstream fin;
    fin.open( fin_name.c_str() );   // convert from string to char [] (C style)
    ofstream fout;
    fout.open( fout_name.c_str() );
    int  income; float tax;
     while ( fin >> income ){
          if( income < cutoff )
            tax = rate1*income;
          else
             tax = rate2*income;
         fout << "Income = " << income << " Drachma \n"
                << "       Tax: " << (int) tax*100.0 << " Lepta" << endl;
     }//while
    fin.close();                          // close file objects
    fout.close();
    return 0;
}
```

Strings

- The standard library ***string*** type provides a variety of useful string operations

```
string s1 = "Hello";
string s2 = "world";
void m1( )
{
    string s3 = s1 + " , " + s2 + "!\n";
    s3 += '\n';
    cout << s3;
}

string name = "Niels Stroustrup";
void m3( )
{
     string s = name.substr(6, 10);  // s = "Stroustrup"
     name.replace(0, 5, "Nicholas"); // name becomes "Nicholas Stroustrup"
}
```

C++ Class string

- String variables and objects can be assigned and concatenated.

```
string r, s, t;
   r = "Hello World";
   s = r;  cout << s << endl;  // prints "Hello World" to the screen
   cin >> t;  //reads a string into variable, t
   r  += t;    //appends t on the right end of r
   r  +=  "What is this World coming to?";  //appends C-string to right
   end of r
   r  += '#';  //appends a character to right end of r
```

- Methods to use with string objects
  - size() yields the length of the string (string::size_type)
  - Length() yields the length of the string (string::size_type)
  - c_str() converts to C-style
  - insert() inserts the operand string at a given position into (*this) string
  - find() searches (*this) for the first occurrence of the operand string
  - substr() returns a substring of (*this) defined by parameter values

Containers

- A class with main purpose of holding an object is commonly called a *container*
  - Much computing involves creating collections of various forms of objects and then manipulating such collections
  - Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program
  - Standard library provides useful containers

Containers - List

- The standard library provides the list type

```
list<Entry> phone_book;
void print_entry (const string& s)
{
     typedef list<Entry>::const_iterator LI;
     for (LI i = phone_book.begin();  i != phone_book.end(); i++){
                Entry& e = *i;    // reference used as shorthand
                if ( s == e.name ) cout << e.name << ‘ ’ << e.number <<
‘\n’;
     }
}
  void add_entry(Entry& e, list<Entry>::iterator I)
  {
       phone_book.push_front(e);  // add at beginning
       phone_book.push_back(e);   // add at end
       phone_book.insert(i, e);   // add before the element’I’ refers to
  }
```

Summary: Standard Containers

- Standard library provides some of the most general and useful container types

vector<T>	A variable-sized vector
list<T>	A doubly-linked list
queue<T>	A queue
stack<T>	A stack
deque<T>	A double-ended queue
priority_queue<T>	A queue sorted by a value
set<T>	A set
multiset<T>	A set in which a value can occur many times
map<key, val>	An associative array
multimap<key, val>	A map in which a value can occur many times

Algorithms

- The standard library provides the most common algorithms for containers
  - For example, the following sorts a vector and places a copy of each unique vector element on a list

```
void  f(list<Entry>& ve, vector<Entry>& le)
{
        sort(ve.begin(), ve.end());
        copy_unique(ve.begin(), ve.end(),
le.begin());
}
```

Algorithms

- Standard Library Algorithms

Selected Standard Algorithms	
for_each( )	Invoke function for each element
find( )	Find first occurrence of arguments
find_if( )	Find first match of predicate
count( )	Count occurrences of element
count_if( )	Count matches of predicate
replace( )	Replace element with new value
replace_if( )	Replace element that matches predicate with new value
copy( )	Copy elements
unique_copy( )	Copy elements that are not duplicates
sort( )	Sort elements
equal_range( )	Find all elements with equivalent values
merge( )	Merge sorted sequences

Iterators

- Definition: *Iterators are objects that enable the programmer to successively access (iterate over) elements stored in a container object without having to know or use the internal data structures and organization used by the container class.*

  In C++, iterators are defined as nested classes within their associated container classes. Furthermore, they have exactly the same properties as pointers to container elements.

- Iterator Applications
  - *Outputting all elements of a container.*
  - *Updating all elements of a container, or all elements satisfying a given condition.*
  - *Searching a container for a given element.*
  - *Deleting or removing all elements satisfying a given condition.*
  - *Sorting the elements in a container.*

Iterators

- Iterator Methods for List Containers

  The following methods operate on *list::iterator and list::reverse_iterator*

  * *(unary dereference operator) gives access to the list element referenced by the iterator.*

  ++ *(postfix increment) advances the iterator to the next list element;*
  (closer to end()); when the iterator is at end(), then ++ advances it to begin()

  -- *(postfix decrement) advances the iterator to the previous list element;*
  (closer to begin()); when the iterator is at begin(), -- advances it to end()

  == *(iterator equality) returns true iff two iterators reference the same list element (the elements themselves may not be equal)*

  != *(iterator not equal) returns true iff two iterators do not reference the same list element.*

- Example

```
std::list<int>      listofint;                  // create a list of integers
std::list<int>::iterator    intiter;            // create an iterator for the list of integers
for( intiter = listofint.begin(); intiter < listofint.end();  intiter++ )
    if ( *intiter < 0 ) cout << *intiter;
```

Math in C++

- How to calculate the square root of a function? We are able to do that using a C++ mathematical library function.
- E.g. the argument to the sqrt function can be either an integer or real value (function overloading).

Expression	Value Returned
sqrt(4)	**2.0**
sqrt (16)	**4.0**
sqrt(6.45)	**2.56**

Math in C++

- To access these functions in a program requires that the mathematical header file named math.h, be included with the function.

- Reminder: This done by the following preprocessor statement at the top of any program using a mathematical function:

```
#include<math.h> // no semicolon
```

- Probably also need –lm compiler flag!

Appendix:
Some C++ Features
C Programmers Should Know

Object-Oriented Idea

- Make all objects, whether C-defined or user-defined, first-class objects
- For C++ structures (called classes) allow:
  – functions to be associated with the class
  – only allow certain functions to access the internals of the class
  – allow the user to re-define existing functions (for example, input and output) to work on class

Classes of Objects in C++

- Classes
  - similar to structures in C (in fact, you can can still use the struct definition)
  - have fields corresponding to fields of a structure in C (similar to variables)
  - have fields corresponding to functions in C (functions that can be applied to that structure)
  - some fields are accessible by everyone, some not (data hiding)
  - some fields shared by the entire class

Inline Functions

- Problems with macros in C
  – Can be a source of problems
  – Has no class scope !!
- C++ solves the problem with inline functions
  – Under the control of the compiler
  – Expanded in-place

Counter Variables in a For Loop

- You can declare the variable(s) used in a for loop in the initialization section of the for loop
  - good when counter used in for loop only exists in for loop (variable is throw-away)
- Example

```
for (int I = 0; I < 5; I++)
  printf("%d\n",I);
```

- Variable exists only during for loop (goes away when loop ends)

Initializing Global Variables

- Not restricted to using constant literal values in initializing global variables, can use any evaluable expression
- Example:

```
int rows = 5;
int cols = 6;
int size = rows * cols;

void main() {
...
```

Initializing Array Elements

- When giving a list of initial array values in C++, you can use expressions that have to be evaluated
- Values calculated at run-time before initialization done
- Example:

```
void main() {
  int n1, n2, n3;
  int *nptr[] = { &n1, &n2, &n3 };
```

void*

- In C it is legal to cast other pointers to and from a void *
- In C++ this is an error, to cast you should use an explicit casting command
- Example:

```
int N;
int *P = &N;
void *Q = P;            // illegal in C++
void *R = (void *) P; // ok
```

C++ Class string

- Access to class *string* requires the following #include statement:

```
#include <string>
using namespace std;
```

- A *string* is an object in C++; strings are completely different types from C-style arrays (char *).
- String variables are initialized by default to the null string (“”).
- String literals are C-style arrays, not members of class string. However, C++ automatically converts from C-style literals to string instances in most of the obvious places, such as variable initializers (see below)

```
string  str = “This is a C-style char array.”;
```

- String objects can be converted to C-style strings (null byte terminated) using the function, c_str().

```
ifstream  fin;
string  filename;
cout << “Enter file name: “;  cin >> filename;
fin.open( filename.c_str(), ios::in );
```

NULL in C++

- C++ does not use the value NULL, instead NULL is always 0 in C++, so we simply use 0
- Example:

```
int *P = 0; // equivalent to
            // setting P to NULL
```

- Can check for a 0 pointer as if true/false:

```
if (!P)  // P is 0 (NULL)
   ...
else // P is not 0 (non-NULL)
   ...
```

Tags and struct

- When using struct command in C++ (and for other tagged types), can create type using tag format and not use tag in variable declaration:

```
struct MyType {
  int A;
  float B;
};
MyType V;
```

enum in C++

- Enumerated types not directly represented as integers in C++
  - certain operations that are legal in C do not work in C++
- Example:

```
void main() {
  enum Color { red, blue, green };
  Color c = red;
  c = blue;
  c = 1;  // Error in C++
  ++c;  // Error in C++
```

Using the C Standard Library

- Access C runtime library by removing ".h" from header files, and prepending "c"

```
#include <cstring>     // strcmp, strlen, etc.
#include <cstdio>      // printf, scanf, etc.
#include <cerrno>      // errno, strerror, etc.
#include <cctype>      // isalnum, isdigit, etc.
#include <cstdlib>     // malloc, free, etc.

// etc.
```

Hybrid C / C++ Programs

- Calling C functions from C++

```
extern "C" void f (int i, char c, float x);
```

- Allow C++ functions to be called from C

```
// This is C++ code
// Declare f(int,char,float) using extern C:
extern "C" void f(int i, char c, float x);
// ...
// Define f(int,char,float) in some C++ module
void f(int i, char c, float x){
   // ...
}
```

Resources and Further Reading

WWW:

http://www.desy.de/gna/html/cc/Tutorial/tutorial.html
http://www.cs.fit.edu/~mmahoney/cse2050/introcpp.html
http://www.acm.org/crossroads/xrds1-1/ovp.html
http://www.thefreecountry.com/compilers/cpp.shtml

Textbook this lecture is based on:

Bjarne Stroustrup, *The C++ Programming Language*,
3rd Ed,, Addison Wesley, 1997.